Experimental Evaluation of a New Shortest Path Algorithm*

Seth Pettie, Vijaya Ramachandran, and Srinath Sridhar
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{seth,vlr,srinath}@cs.utexas.edu

UTCS TR-01-37

December 11, 2001

Abstract

We evaluate the practical efficiency of a new shortest path algorithm for undirected graphs which was developed by the first two authors. This algorithm works on the fundamental *comparison-addition model*.

Theoretically, this new algorithm out-performs Dijkstra's algorithm on sparse graphs for the all-pairs shortest path problem, and more generally, for the problem of computing single-source shortest paths from $\omega(1)$ different sources. Our extensive experimental analysis demonstrates that this is also the case in practice. We present results which show the new algorithm to run faster than Dijkstra's on a variety of sparse graphs when the number of vertices ranges from a few thousand to a few million, and when computing single-source shortest paths from as few as three different sources.

1 Introduction

The shortest paths problem on graphs is one of the most widely-studied combinatorial optimization problems. Given an edge-weighted graph, a path from a vertex u to a vertex v is a *shortest path* if its total length is minimum among all u-to-v paths. The complexity of finding shortest paths seems to depend upon how the problem is formulated and what kinds of assumptions we place on the graph, its edge-lengths and the machine model. Most shortest path algorithms for graphs can be well-categorized by the following choices.

1. Whether shortest paths are computed from a single *source* vertex to all other vertices (SSSP), or between all pairs of vertices (APSP). One should also consider the intermediate problem of computing shortest paths from *multiple* specified sources (MSSP).
2. Whether the edge lengths are non-negative or arbitrary.
3. Whether the graph is directed or undirected.
4. Whether shortest paths are computed using just *comparison & addition* operations, or whether they are computed assuming a specific edge-length representation (typically integers in binary) and operations specific to that representation. Comparison-addition based algorithms are necessarily general and they work when edge-lengths are either integers or real numbers.

*This work was supported by Texas Advanced Research Program Grant 003658-0029-1999 and NSF Grant CCR-9988160. Seth Pettie was also supported by an MCD Graduate Fellowship.

There is a wealth of literature on variations of the shortest path problem,[1] however despite such intense research, very few of the results beyond the classical algorithms of Dijkstra, Bellman-Ford, Floyd-Warshall, and min-plus matrix multiplication [AHU74, CLR90] work with real-valued edge-lengths using only comparisons and additions.[2]

Previous experimental studies of shortest path algorithms [CGR96, GS97, G01b] focussed on very restricted classes of inputs, where the edge lengths were assumed to be uniformly distributed, relatively small integers. This approach may be preferable for a specific application, however any algorithm implemented for more general use must be *robust*. By robust we mean that it makes no assumptions on the distribution of inputs, and minimal assumptions on the *programming interface* to the input (in the case of shortest path problems this leads naturally to the comparison-addition model); we elaborate on this in Section 2. A fact which many find startling is that Dijkstra's 1959 algorithm is still the best robust SSSP & APSP algorithm for positively-weighted sparse directed graphs.

In this paper we evaluate the performance of the recent *undirected* shortest path algorithm of Pettie & Ramachandran [PR02], henceforth the *PR algorithm*. The PR algorithm is a robust, comparison-addition based algorithm for solving undirected SSSP from multiple specified sources (MSSP). It works by precomputing a certain structure called the 'component hierarchy', or *CH* (first proposed by Thorup [Tho99], for use with integer edge lengths) in time $O(m+n \log n)$. Once the CH is constructed SSSP is solved from any source in $O(m\alpha(m,n))$ time, where α is the very slow-growing inverse-Ackermann function. Theoretically this algorithm is asymptotically faster than Dijkstra's when the number of sources is $\omega(1)$ and the number of edges is $o(n \log n)$.

The PR algorithm (as well as [Tho99, Hag00]) can also tolerate a dynamic graph in some circumstances. If a component hierarchy is constructed for a graph G, SSSP can be solved in $O(m\alpha(m,n))$ time on any graph G' derived from G by altering each edge weight by up to a constant factor.

As mentioned above, there are only a few shortest path algorithms that work on the comparison-addition model, and there is only one robust algorithm in direct competition with PR, namely Dijkstra's. The Bellman-Ford, Floyd-Warshall, and min-plus matrix multiplication algorithms handle negative edge lengths and as a consequence are considerably less efficient than the PR algorithm (quadratic time for SSSP and cubic for APSP). The fastest implementation [Tak92] of Fredman's algorithm [F76] for APSP also takes almost cubic time. The average-case algorithms in [KKP93, McG91, Jak91, MT87] only provide improvements on very dense random graphs.

We evaluate the practical efficiency of the PR algorithm for the MSSP problem on undirected graphs by comparing it with Dijkstra's algorithm. The MSSP problem generalizes the SSSP-APSP extremes, and could be more relevant in some practical scenarios. For instance, a recent algorithm of Thorup [Tho01] for the graphic facility location and k-median problems performs SSSP computations from a polylog number of sources. Our experiments indicate quite convincingly that the Pettie-Ramachandran algorithm outperforms Dijkstra on sparse graphs when computing SSSP from a sufficient number of sources, as few as 3 or 4 in several cases. We obtained this result across all classes of sparse graphs that we considered except for the so-called 'long grids' [CGR96]. We also compare the PR algorithm to breadth first search, a natural lower bound on SSSP and a useful routine to normalize the running times of shortest path algorithms across different architectures. We elaborate on this and other aspects of our results in Section 6. Clearly, our results also apply to the APSP problem, and they show that the PR algorithm outperforms Dijkstra's algorithm for the APSP problem on sparse graphs.

The rest of the paper is organized as follows. In Section 2 we delineate the scope of our study. In Section 3 we give an overview of Dijkstra's algorithm and the PR algorithm. Section 4 describes the design choices we made in implementing the two algorithms. Section 5 describes our experimental set-up, and Section 5.1 the types of graphs we used. Section 6 provides our results. Section 7 ends with a discussion.

[1] For an up-to-date survey of shortest path algorithms, see Zwick [Z01] (an updated version is available on-line).

[2] Some exceptions to this rule are Fredman's min-plus matrix multiplication algorithm [F76] and several algorithms with good *average-case* performance: [MT87, KKP93, KS98, Mey01, G01]

2 Scope of this Work

The focus of this paper is *robust* shortest path algorithms, so it is worthwhile to state here exactly what we mean by the term. A robust shortest path algorithm should be robust with respect to:

Input format. The algorithm should work with minimal assumptions on the input format and the programming "hooks" to manipulate the input. The assumption that edge-lengths are subject to comparison and addition operations is minimal since these operations are both necessary and sufficient to solve shortest path problem.

Graph type. The algorithm should work well on *all* graph sizes & topologies. It should not depend on the graph being overly structured (e.g. grids) or overly random (e.g. the $G_{n,m}$ distr.).

Edge-length distribution. The algorithm should not be adversely affected by the range or distribution on edge-lengths, nor should it depend upon the edge-lengths being chosen independently at random.

Some may object to the first criterion because, at some level, edge lengths are represented as `int`s or `double`s; one might as well assume such an input. This is not quite true. For instance, the LEDA platform [MN99] uses different types for rationals, high-precision floating point numbers, and 'real' numbers with provable accuracy guarantees, and Java has similar types BigDecimal and BigInteger. A robust algorithm can be used with all such types with little or no modification, and can be ported to different platforms with minimal modifications.

The bottom line is that robust algorithms are fit for use in a general setting where the format and distribution of inputs is unknown and/or varies. Nothing precludes the use of other specialized shortest path algorithms (indeed, those tailored to small integer weights, e.g. [GS97], will likely be faster), however, depending solely on such an algorithm is clearly unwise.

In our experiments we focus primarily on classes of *sparse graphs*, which we define as having an edge-to-vertex ratio less than $\log n$. Sparse graphs frequently arise naturally; e.g. all planar and grid-like graphs are sparse, and the evidence shows the 'web graph' also to be sparse. Denser graphs are important as well, but as a practical matter the SSSP problem has essentially been solved: Dijkstra's algorithm runs in linear time for densities greater than $\log n$. The "sorting bottleneck" in Dijkstra's algorithm is only apparent for sparse graphs.

3 Overview of the algorithms

Dijkstra's algorithm [Dij59] for SSSP (see [CLR90] or [AHU74]) visits the vertices in order of increasing distance from the source. It maintains a set S of visited vertices whose distance from the source has been established, and a tentative distance $D(v)$ to each unvisited vertex v. $D(v)$ is an upper bound on the actual distance to v, denoted $d(v)$; it is the length of the shortest path from the source to v in the subgraph induced by $S \cup \{v\}$. Dijkstra's algorithm repeatedly finds the unvisited vertex with minimum tentative distance, adds it to the set S and updates D-values appropriately.

Rather than giving a description of the Pettie-Ramachandran [PR02] algorithm (which is somewhat involved), we will instead describe the component hierarchy *approach* put forward by Thorup [Tho99]. Suppose that we are charged with finding all vertices within distance b of the source, that is, all v such that $d(v) \in [0, b)$. One method is to run Dijkstra's algorithm (which visits vertices in order of their d-value) until a vertex with d-value outside $[0, b)$ is visited. Thorup observed that if we choose $t < b$ and find the graph G_t consisting of edges shorter than t, the connected components of G_t, say $\mathcal{G}_t$, can be dealt with separately in the following sense. We can simulate which vertices Dijkstra's algorithm *would* visit for each connected component in $\mathcal{G}_t$, first over the interval $[0, t)$, then $[t, 2t)$, $[2t, 3t)$, up to $[\lfloor \frac{b}{t} \rfloor t, b)$. It is shown in [Tho99] (see also [PR02]) that these separate subproblems do not "interfere" with each other in a technical sense. The subproblems generated by Thorup's approach are solved recursively. The component hierarchy is a rooted tree which represents how the graph is decomposed; it is determined by the underlying graph and choices of t made in the algorithm. The basic procedure in component hierarchy-based algorithms [Tho99, Hag00, PR02] is `Visit`(x, I), which takes a component hierarchy node x and an interval I, and visits all vertices in the subgraph corresponding to x whose d-values lie in I.

4 Design Choices

4.1 Dijkstra's Algorithm

We use a *pairing heap* [F+86] to implement the priority queue in Dijkstra's algorithm. We made this choice based on the results reported in [MS94] for minimum spanning tree (MST) algorithms. In that experiment the pairing heap was found to be superior to the Fibonacci heap (the choice for the theoretical bound), as well as d-ary heaps, relaxed heaps and splay heaps in implementations of the Prim-Dijkstra MST algorithm.[3] Since the Prim-Dijkstra MST algorithm has the same structure as Dijkstra's SSSP algorithm (Dijkstra presents both of these algorithms together in his classic paper [Dij59]), the pairing heap appears to be the right choice for this algorithm.

The experimental studies by Goldberg [CGR96, GS97, G01b] have used buckets to implement the heap in Dijkstra's algorithm. However, the bucketing strategy they used applies only to integer weights. The bucketing strategies in [Mey01, G01] could apply to arbitrary real edge weights, but they are specifically geared to good performance on edge-weights uniformly distributed in some interval. The method in [G01] can be shown to have bad performance on some natural inputs.[4] In contrast we are evaluating robust, general-purpose algorithms that function in the comparison-addition model.

We experimented with two versions of Dijkstra's algorithm, one which places all vertices on the heap initially with key value ∞ (the traditional method), and the other that keeps on the heap only vertices known to be at finite distance from the source. For sparse graphs one would expect the heap to contain fewer vertices if the second method is used, resulting in a better running time. This is validated by our experimental data. The second method out-performed the first one in all graphs that we tested, so we report results only for the second method.

4.2 Pettie-Ramachandran Algorithm

The primary consideration in [PR02] was asymptotic running time. In our implementation of this algorithm we make several simplifications and adjustments which are more practical but may deteriorate the worst-case asymptotic performance of the algorithm.

1. Finding MST:

   The [PR02] algorithm either assumes the MST is found in $O(m + n \log n)$ time (for the multi-source case) or, for the single source case, in optimal time using the algorithm of [PR00]. Since, for multiple sources, we both find and sort the MST edges, we chose to use Kruskal's MST algorithm, which runs in $O(m \log n)$ time but does both of these tasks in one pass. Some of our data on larger and denser graphs suggests that it may be better to use the Prim-Dijkstra MST algorithm, which is empirically faster than Kruskal's [MS94], followed by a step to sort only the MST edges.

2. Updating D-values:

   In [PR02] the D-value of an internal CH node is defined to be the minimum D-value over its descendant leaves. As leaf D-values change, the internal D-values must be updated. Rather than use Gabow's near-linear time data structure [G85], which is rather complicated, we use the naïve method. Whenever a leaf's D-value decreases, the new D-value is propagated up the CH until an ancestor is reached with an even lower D-value. The worst-case time for updating a D-value is clearly the height of CH, which is $\log R$, where R is the ratio of the maximum to minimum edge-weight; on the other hand, very few ancestors need to be updated in practice.

3. Using Dijkstra on small subproblems:

   The stream-lined nature of Dijkstra's algorithm makes it the preferred choice for computing shortest paths on small graphs. For this reason we revert to Dijkstra's algorithm when the problem size becomes sufficiently small. If `Visit`(x, I) is called on a CH node x with fewer than ν descendant leaves, we run

[3] This algorithm was actually discovered much earlier by Jarník [Jar30].

[4] For instance, where each edge length is chosen independently from one of two uniform distributions with very different ranges.

Dijkstra's algorithm over the interval I rather than calling `Visit` recursively. For *all* the experiments described later, we set $\nu = 50$.

4. Heaps vs. Lazy Bucketing:

   The [PR02] algorithm implements a priority queue with a *comparison-addition based* 'lazy bucketing' structure. This structure provides asymptotic guarantees, but for practical efficiency we decided to use a standard pairing heap to implement the priority queue, augmented with an operation called *threshold* which simulates emptying a bucket. A call to *threshold*(t) returns a list of all heap elements with keys less than t. It is implemented with a simple DFS of the pairing heap. An upper bound on the time for threshold to return k elements is $O(k \log n)$, though in practice it is much faster.

5. Additional Processing of CH:

   In [PR02, Sections 3 & 4] the CH undergoes a round of refinement, which is crucial to the asymptotic running time of the algorithm. We did not implement these refinements, believing their real-world benefits to be negligible. However, our experiments on *hierarchically structured graphs* (which, in effect, have pre-refined CHs) are very encouraging. They suggest that the refinement step could speed up the computation of shortest paths, at the cost of more pre-computation.

4.3 Breadth First Search

We compare the PR algorithm not only with Dijkstra's, but also with breadth first search (BFS), an effective lower bound on the SSSP problem. Our BFS routine is implemented in the usual way, with a FIFO queue [CLR90]. It finds a shortest path (in terms of number of edges) from the source to all other vertices, and computes the lengths of such paths.

5 Experimental Set-up

Our main experimental platform was a SunBlade with a 400 MHz clock and 2GB DRAM and a small cache (.5 MB). The large main memory allowed us to test graphs with millions of vertices. For comparison purposes we also ran our code on selected inputs on the following machines.

1. PC running Debian Linux with a 731 MHz Pentium III processor and 255 MB DRAM.
2. SUN Ultra 60 with a 400 MHz clock, 256 MB DRAM, and a 4 MB cache.
3. HP/UX J282 with 180 MHz clock, 128 MB ECC memory.

5.1 Graph Classes

We ran both algorithms on the following classes of graphs.

$G_{n,m}$. The distribution $G_{n,m}$ assigns equal probability to all graphs with m edges on n labeled vertices (see [ER61, Bo85] for structural properties of $G_{n,m}$). We assign edge-lengths identically and independently, using either the uniform distribution over $[0, 1)$, or the *log-uniform* distribution, where edge lengths are given the value 2^q, q being uniformly distributed over $[0, C)$ for some constant C. We use $C = 100$.

Geometric graphs. Here we generate n random points (the vertices) in the unit square and connect with edges those pairs within some specified distance. Edge-lengths correspond to the distance between points. We present results for distance $1.5/\sqrt{n}$, implying an average degree $\approx 9\pi/4$ which is about 7.

Very sparse graphs. These graphs are generated in two stages: we first generate a random spanning tree, to ensure connectedness, then generate an additional $n/10$ random edges. All edges-lengths are uniformly distributed.

Grid graphs. In many situations the graph topology is not random at all but highly predictable. We examine two classes of grid graphs: $\sqrt{n} \times \sqrt{n}$ square grids and $16 \times n/16$ long grids, both with uniformly distributed edge-lengths [CGR96].